Debugging Declarativo basado en Revisión de Creencias

Claudio A. Vaucheret
Departamento de Informática y Estadística
Universidad Nacional del Comahue

Guillermo R. Simari
Departamento de Ciencias de la Computación
Universidad Nacional del Sur[1]

e-mail: cvaucher@uncoma.edu.ar, grs@criba.edu.ar

Palabras clave: Debugging Declarativo, Teoría de Cambio de Creencias, Programación Lógica

Resumen

En este trabajo, presentamos un procedimiento de debugging basado en una operación de cambio de creencias, que requiere una mínima interacción con el programador. El significado pretendido del programa se incluye en el mismo programa por medio de reglas de oráculo y la operación de cambio de creencias hace que el programa se vuelva compatible con dicha información. El procedimiento utiliza el operador de contracción presentado en [Vaucheret 98] para eliminar las reglas incorrectas del programa y también proveer información adicional para los átomos no cubiertos. La clase de programas lógicos considerados son los programas lógicos no esquemáticos, es decir proposicionales, con negación por falla y extendidos con la negación clásica.

1 Introducción

El debugging declarativo de programas [Shapiro 82, Pereira 86, Dershowitz 87, Calejo 91, Ducassé 93] es un proceso de depuración de errores donde el programador solo necesita un conocimiento declarativo del problema sin necesidad de considerar los aspectos procedurales del comportamiento del sistema. La ventaja comparativa de la programación lógica debida a su carácter declarativo se vería limitada si en la etapa de la corrección de errores el programador debe conocer todo acerca del comportamiento computacional del lenguaje de programación. Por lo tanto los sistemas de debugging declarativo se vuelven un componente muy importante en los sistemas de programación lógica.

Los debuggers declarativos realizan un proceso de búsqueda siguiendo la pista hasta encontrar el error, realizando una constante interacción con el programador para consultar el significado pretendido del programa. El programador juega el rol de un oráculo que conoce el significado pretendido y es consultado con el fin de que el debugger reconozca las reglas incorrectas o los átomos no cubiertos por las reglas del programa.

La investigación en debugging declarativo busca lograr la mayor automatización posible por medio de técnicas que logren reducir la interacción con el programador. En este trabajo, nosotros

[1] Miembro de GIIA (Grupo de Investigación en Inteligencia Artificial) e ICIC (Instituto de Ciencias e Ingeniería de Computación), UNS, Bahía Blanca

proponemos un procedimiento de debugging basado en una operación de cambio de creencias, que no requiere interacción con el programador. El significado pretendido del programa se incluye en el mismo por medio de reglas de oráculo y la operación de cambio de creencias hace que el programa se vuelva compatible con dicha información.

Un programa lógico puede ser pensado como una teoría lógica cuyo modelo describe el funcionamiento de un programa. El modelo pretendido por el programador puede diferir del modelo real, por lo tanto, el objetivo del debugging es extraer esos axiomas de la teoría que tienen la responsabilidad de la discrepancia. Considerado un programa como una teoría lógica, este proceso de debugging se puede considerar una operación de cambio de creencias.

La organización del trabajo es la siguiente, en la sección 2 se revisa el operador de contracción para programas lógicos. Este operador, es luego utilizado en la sección 3 para presentar el procedimiento de debugging.

2 Contracción de Programas Lógicos

En esta sección daremos una revisión del operador de contracción definido en [Vaucheret 98] para programas lógicos con negación clásica y negación por falla. Este operador de contracción está basado en la *safe contraction* definida por Carlos Alchourrón y David Makinson en [Alchourrón 85] para estados epistémicos representados por teorías lógicas. La idea principal del operador de contracción definido en esta sección es que con el fin de contraer un literal de las consecuencias de un programa, en primer lugar se recolectan todos los elementos del programa —reglas y suposiciones de negación por falla— necesarios para inferir dicho literal. A cada conjunto minimal de estos elementos, lo llamaremos una *explicación* del literal. La operación de contracción elimina los elementos de menor valor epistémico de estas explicaciones, eliminando reglas del programa o agregando átomos para invalidar alguna suposición de negación por falla.

Recordemos que un *elemento de regla* es un literal posiblemente precedido por el símbolo de negación por falla *not*. Un elemento de regla L y el correspondiente precedido por el operador *not*, $not\ L$ serán llamados literales complementarios. Con $Comp(L)$ notaremos el literal complementario de L, es decir $Comp(notL) = L$ y $Comp(L) = not\ L$.

Un programa lógico es definido como un conjunto de reglas cuyos cuerpos están formados por elementos de regla. Así, cada regla es de la forma:

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.$$

Todo programa tiene un significado asociado por una semántica determinada. Dicha semántica puede ser bi-valuada como la semántica de modelos estables [Gelfond 88] o tri-valuada como la semántica bien fundada [Van Gelder 91]. Los programas lógicos que trataremos aquí cuentan con dos tipos de negación, la negación por falla y la negación clásica $\neg$, esta clase de programas es llamada la clase de programas lógicos extendidos y el significado de estos programas estará dado por semánticas extendidas que pueden ser también bi-valuadas[Gelfond 90] o tri-valuadas [Przymusinski 90] [Pereira 92].

Para el propósito de este trabajo, los programas lógicos serán considerados teorías lógicas por lo que le asignaremos un operador de consecuencia que estará basado en alguna semántica de programas lógicos. Para el caso general consideraremos una interpretación I de un programa Π como un par $\langle V, F\rangle$ donde V y F son subconjuntos disjuntos de los literales del programa, el conjunto V contiene todos los literales verdaderos en I, el conjunto F contiene los literales falsos en I y el valor de verdad del resto de los literales es desconocido o indefinido. Dada una interpretación de este tipo que asigna un significado a un programa, las consecuencias de un programa se definen de la siguiente manera:

Definición 2.1 *Sea $\langle V, F\rangle$ una interpretación trivaluada que asigna un significado a el programa Π y sea L un literal, entonces $L \in Cn(\Pi)$ si y solo si $L \in V$ y $not\ L \in Cn(\Pi)$ si y sólo si $L \in F$.*

Ejemplo 1 *Consideremos el siguiente programa* Π *y para definir su significado utilizaremos la semántica bien fundada de programas lógicos.*

$$p \leftarrow \neg r, not\ q.$$
$$p \leftarrow not\ b.$$
$$\neg r \leftarrow a.$$
$$b \leftarrow q, c.$$
$$a.$$

Dado que el modelo asignado por la semántica es $\langle\{a, p, \neg r\}, \{c, b, q\}\rangle$*, es decir,* a *,* p *y* $\neg r$ *son consecuencias bien fundadas y* c*,* b *y* q *son infundados, entonces las consecuencias de* Π *es el conjunto* $Cn(\Pi) = \{a, \neg r, not\ c, not\ q, not\ b, p\}$

2.1 Explicaciones de un Literal

Intuitivamente, la explicación de un literal, será aquella porción del programa que contribuye en inferir dicho literal. La explicación de un literal dentro de un programa lógico, es el equivalente al kernel de una sentencia en la teoría de Cambio de creencias [Alchourrón 85]. Para el caso de programas lógicos, una explicación de un literal es algo más compleja que un subconjunto del programa. Debido al operador de negación por falla, un programa lógico puede inferir literales basado en suposiciones, es decir dicho operador permite que ante la ausencia de una información el programa pueda inferir literales. Por lo tanto una explicación de un literal en el contexto de un programa debe estar estructurado por una porción del programa, sumado a un conjunto de suposiciones constituido por información que el programa no debe inferir.

La definición formal de una explicación para el caso de programas con negación por falla es la siguiente:

Definición 2.2 *Dado un programa* Π *y un literal* L *una* explicación *en* Π *es un triple* $\langle I, O, L\rangle$ *minimal donde se cumplen las siguientes condiciones:*

- $I \subseteq \Pi$*,* $O \subseteq Lit(\Pi)$*,*
- $L \in Cn(I)$*,*
- $L \notin Cn(I \cup O)$
- *para todo* $x \in O$ *se cumple que not* $x \in Cn(\Pi)$
- *para todo* $x \in O$ *se cumple que not* $x \in Cn(\Pi \cup \{Lit(\Pi) - O\})$

Llamamos $Lit(\Pi)$ a todos los literales clásicos que ocurren en alguna regla de Π. La explicación es minimal, en el sentido de que no existe otro triple $\langle I', O', L\rangle$ que cumpla las condiciones de arriba y $I' \subset I$ y $O' \subset O$. El componente I de una explicación contiene las reglas del programa necesarias para inferir L y el componente O contiene los elementos que no deben estar en el significado del programa de manera de poder inferir L al que llamamos el consecuente de la explicación. De esta manera, una explicación se puede invalidar con la ausencia de un elemento de I o con la presencia de un elemento de O en el significado del programa.

Ejemplo 2 *Si tomamos el programa del ejemplo 1 con su significado, las siguientes son ejemplos de explicaciones en* Π *para el literal* p*:*

- $\langle\{p \leftarrow \neg r, not\ q; \neg r \leftarrow a; a\}, \{q\}, p\rangle$
- $\langle\{p \leftarrow not\ b\}, \{q; b\}, p\rangle$
- $\langle\{p \leftarrow not\ b\}, \{c; b\}, p\rangle$

Teniendo en cuenta un programa y su significado, se puede definir un procedimiento que obtenga las explicaciones de un literal dado. Dicho procedimiento es el siguiente:

Definición 2.3

entrada: *un programa* Π , $Cn(\Pi)$ *sus consecuencias y* L *un elemento de regla.*
salida: *una explicación* $e = \langle I, O, L\rangle$ *en* Π

Si L *es un literal:*
Se Selecciona una regla $L \leftarrow B_1, \ldots, B_n$ *donde cada* $B_i \in Cn(\Pi)$
Se asigna a e *el triple* $\langle I, O, L\rangle$
inicializado con los siguientes valores
$I = \{L \leftarrow B_1, \ldots, B_n\}$ *y* $O = \emptyset$
para cada B_i*:*
se obtine recursivamente una explicación $\langle I'_i, O'_i, B_i\rangle$,
se asigna $I \leftarrow I \cup I'_i$ *y* $O \leftarrow O \cup O'_i$
Retorna e

Si L *es un elemento de regla de la forma not* A*:*
Se Selecciona de cada regla $A \leftarrow B_1, \ldots, B_n$, *un elemento*
de su cuerpo B_j *tal que* $B_j \notin Cn(\Pi)$
se asigna a e *el triple* $\langle I, O, L\rangle$
inicializado con los siguientes valores
$I = \emptyset$ *y* $O = \{A\}$
para cada B_j*:*
se obtiene recursivamente una explicación $\langle I'_j, O'_j, Comp(B_j)\rangle$,
se asigna $I \leftarrow I \cup I'_j$ *y* $O \leftarrow O \cup O'_j$
Retorna e
Si no existe ninguna regla con A *en la cabeza retorna* $e = \langle\{\}, \{A\}, L\rangle$

El procedimiento para encontrar una explicación $\langle I, O, L\rangle$ busca una regla que contribuya en inferir L, es decir una regla cuya cabeza sea L y todos los literales del cuerpo pertenezcan al significado del programa, dicha regla pasa a formar parte del componente I de la explicación. Luego esta explicación debe completarse con una explicación para cada literal del cuerpo. Los componentes de estas explicaciones se agregan respectivamente a los componentes I y O de la explicación para L. Para el caso de que L sea un literal default, es decir *not* A, Todas las reglas con A en la cabeza están inactivas por algún elemento que no pertenece al significado del programa. El procedimiento construye una explicación para L formada por las explicaciones de los complementos de dichos elementos.

Las consecuencias de un programa están justificadas por la existencia de sus explicaciones:

Proposición 2.4 *Sea* L *un elemento de regla y* Π *un programa consistente, entonces,* $L \in Cn(\Pi)$ *si y solo si existe por lo menos una explicación* $\langle I, O, L\rangle$ *en* Π.

Prueba:

Veamos por casos en primer lugar si L es un literal, supongamos que existe una explicación $\langle I, O, L\rangle$ en Π, sea $L \leftarrow B_1, \ldots, B_n$ la primer regla seleccionada por el procedimiento de la definición 2.3. Dado que la regla fue seleccionada, $\{B_1, \ldots, B_n\} \subseteq Cn(\Pi)$. Sea $I = \langle V, F\rangle$ la interpretación asociada a $Cn(\Pi)$ por la definición 2.1. Dado que I es modelo de Π esta regla determina que $L \in V$ y por lo

tanto $L \in Cn(\Pi)$. Si L en cambio es un elemento de regla de la forma *not* A, entonces como I es modelo mínimo, $A \in F$ y por lo tanto por la definición 2.1 *not* $A \in Cn(\Pi)$.

Recíprocamente, sea $L \in Cn(\Pi)$, y supongamos que el procedimiento 2.3 no obtenga una explicación, por lo tanto sea M un elemento de regla (puede ser L o algún B_i en alguna instancia recursiva) tal que $M \in Cn(\Pi)$ y el procedimiento falle en encontrar una explicación. Existen dos casos:

- M es un literal clásico y no existe ninguna regla $M \leftarrow B_1, \ldots, B_n$ donde cada $B_i \in Cn(\Pi)$. Pero si este es el caso, entonces $M \in F$, luego $M \notin V$ y contradice el hecho de que $M \in Cn(\Pi)$.
- M es un elemento de regla de la forma *not* A y existe una regla $A \leftarrow B_1, \ldots, B_n$ donde cada $B_i \in Cn(\Pi)$, Pero si este es el caso, entonces $A \in V$, luego $A \notin F$, y contradice el hecho de que *not* $A \in Cn(\Pi)$.

Por lo tanto existe una explicación con consecuente L.

A continuación veamos ejemplos de explicaciones de elementos de regla de programas lógicos.

Ejemplo 3 *Tomando el programa* Π *y sus consecuencias del ejemplo 1, los siguientes son ejemplos de explicaciones calculadas por el procedimiento anterior:*

- $\langle \{a.\}, \{\}, a \rangle$
  dado que $a.$ es la unica regla para a

- $\langle \{\neg r \leftarrow a; a\}, \{\}, \neg r \rangle$
  como $a \in Cn(\Pi)$ se selecciona la regla $r \leftarrow a$. Primero se asigna a e,

  $$e = \langle \{\neg r \leftarrow a.\}, \{\}, \neg r \rangle$$

  y luego de obtener la explicación de a el procedimiento devuelve

  $$e = \langle \{\neg r \leftarrow a; a\}, \{\}, \neg r \rangle$$

- $\langle \{\}, \{q\}, not\ q \rangle$
  dado que no hay reglas para q

- $\langle \{\}, \{b, q\}, not\ b \rangle$
  De la única regla para b, se selecciona q ya que $q \notin Cn(\Pi)$. El procedimiento primero asigna a e,

  $$e = \langle \{\}, \{b\}, not\ b \rangle$$

  y luego de obtener la explicación de *not* q devuelve

  $$e = \langle \{\}, \{b, q\}, not\ b \rangle$$

- $\langle \{\}, \{b, c\}, not\ b \rangle$
  igual que en el caso anterior pero seleccionando $c \notin Cn(\Pi)$

- $\langle \{p \leftarrow \neg r, not\ q; \neg r \leftarrow a; a\}, \{q\}, p \rangle$
  seleccionando la primera regla para p provisto que *not* q y $\neg r$ pertenecen a $Cn(\Pi)$ el procedimiento sucesivamente asigna a e
  $e = \langle \{p \leftarrow \neg r, not\ q\}, \{\}, p \rangle$
  $e = \langle \{p \leftarrow \neg r, not\ q\}, \{q\}, p \rangle$ al obtener la explicación de *not* q
  $e = \langle \{p \leftarrow \neg r, not\ q; \neg r \leftarrow a; a\}, \{q\}, p \rangle$ al obtener la explicación de $\neg r$

- $\langle \{p \leftarrow not\ b\}, \{q, b\}, p\rangle$
  seleccionando la segunda regla para p provisto que $not\ b$ pertenece a $Cn(\Pi)$ el procedimiento sucesivamente asigna a e
  $e = \langle \{p \leftarrow not\ b\}, \{\}, p\rangle$
  $e = \langle \{p \leftarrow not\ b\}, \{b, q\}, p\rangle$ al obtener la explicación de $not\ b$

Con el fin de realizar una operación de contracción de un programa Π por un literal L, se debe herir cada una de las explicaciones de Π cuyo consecuente sea L. El programa resultante de la operación de contracción $\Pi \div L$ no debe contener explicaciones con L como consecuente. La forma de invalidar una explicación $\langle I, O, L\rangle$ es *revisar* alguno de los elementos de I o de O. En el primer caso $\Pi \div L$ no debe contener algún elemento de I, y en el segundo para algún elemento x de O $not\ x$ no debe pertenecer a $Cn(\Pi \div L)$.

En lugar de considerar a todas las reglas de un programa o sus suposiciones por negación por falla como revisables se pueden clasificar los elementos de un programa distinguiendo aquellos que uno desea proteger de la revisión y aquellos que son revisables. En la siguiente subsección definiremos teniendo en cuenta esta aproximación, las explicaciones revisables y las revisiones para un programa.

2.2 Explicaciones Revisables y Revisiones

Cuando para un literal L no hay reglas en un programa Π, $not\ L$ es inferido por el programa debido a la suposición de mundo cerrado dado por la negación por falla. En el momento de decidir que elementos de una explicación se deben revisar en una operación de contracción sería coherente considerar primero estas suposiciones realizadas por la negación por falla que por ejemplo alguna regla del programa. Dentro de las cláusulas de un programa, los hechos pueden ser considerados mas revisables que las reglas. Este criterio es similar a la actualización de base de conocimientos, donde el conocimiento particular es más revisable que el conocimiento general. En este caso los hechos serían un conocimiento particular de una aplicación y las reglas de un programa se considerarían un conocimiento más general. Para distinguir los elementos de un programa factibles de revisión introduciremos la definición de *revisables* de un programa.

Definición 2.5 *Dado un programa* Π *los* revisables *de* Π $Rev(\Pi)$ *es un par* $\langle \mathcal{I}, \mathcal{O}\rangle$ *donde* $\mathcal{I} \subseteq \Pi$ *y si* $x \in \mathcal{O}$ *entonces not* $x \in Cn(\Pi)$

Ejemplo 4 *Si consideramos por ejemplo el criterio de tomar a los hechos de un programa y sus suposiciones por negación por falla, los elementos revisables del programa* Π *del ejemplo 1 son:*

$$Rev(\Pi) = \langle \{a\}, \{q, c\}\rangle$$

a es el único hecho del programa y los literales q y c, no tienen reglas en Π

Dejando establecido que no todos los elementos de un programa son revisables, surge la posibilidad de que la operación de contracción no pueda realizarse para algún literal dado. La definición de revisables de un programa nos permite definir un operador de semi-contracción, es decir que la contracción $\Pi \div L$ se realizará solo si se pueden invalidar todas las explicaciones de L.

Definición 2.6 *Dado un programa* Π *donde* $Rev(\Pi) = \langle \mathcal{I}, \mathcal{O}\rangle$. *Sea* $e = \langle I, O, L\rangle$ *una explicación en* Π. *Si* $\mathcal{I} \cap I = \emptyset$ *y* $\mathcal{O} \cap O = \emptyset$ *entonces la explicación e es no revisable. En caso contrario, e es revisable.*

Si ninguno de los elementos de una explicación es revisable, entonces dicha explicación no se puede invalidar en una operación de contracción. Esto nos permite definir cuándo se puede aplicar una operación de contracción a un programa. Para poder contraer un programa Π por L, todas las explicaciones $\langle I, O, L\rangle$ del programa, deben ser revisables, porque si alguna explicación se mantuviera vigente, L seguiría perteneciendo al significado de Π.

Definición 2.7 *Un programa* Π *se puede contraer por* L *si todas las explicaciones de* Π *con consecuente* L *son revisables.*

Para la operación de contracción nos interesa considerar solo los elementos revisables. Definiremos entonces un tipo de explicaciones que contenga solo los elementos plausibles de revisión.

Definición 2.8 *Dado un programa* Π *donde* $Rev(\Pi) = \langle \mathcal{I}, \mathcal{O} \rangle$. *Sea* $e = \langle I, O, L \rangle$ *una explicación en* Π. *Si* e *es revisable, entonces* $\langle \mathcal{I} \cap I, \mathcal{O} \cap O, c \rangle$ *es un* soporte revisable *en* Π.

Un soporte revisable $\langle I, O, L \rangle$ de Π contiene componentes revisables del programa Π que contribuyen a que L pertenezca a $Cn(\Pi)$. El concepto de soporte revisable corresponde al de kernel para una sentencia en la definición de contracción kernel en la teoría de cambio de creencias [Hansson 94].

Ejemplo 5 *Sea* Π *el programa del ejemplo 1, suponga* $Rev(\Pi) = \langle \{a\}, \{q, c\} \rangle$. *todas las explicaciones de* p *son revisables y los soportes revisables son:*

- $\langle \{a\}, \{q\}, p \rangle$
- $\langle \{\}, \{q\}, p \rangle$
- $\langle \{\}, \{c\}, p \rangle$

Dado un programa Π y un literal L, vamos a definir una revisión de L en ese programa como un conjunto de elementos de Π que deben ser removidos con el fin de contraer L. Por lo tanto dicha revisión debe contener elementos de cada uno de los soportes revisables con consecuente L.

Definición 2.9 *Sea* Π *un programa contraíble por* L, *una* revisión *es un triple* $\langle I, O, L \rangle$ *minimal tal que para todo soporte revisable en* Π $\langle I', O', L \rangle$ *se cumple ya sea que* $I \cap I' \neq \emptyset$ *o* $O \cap O' \neq \emptyset$

Una revisión es una selección de elementos de un programa a ser retirados con el fin de realizar la operación de contracción.

2.3 Seleccionando una Revisión

Con el objetivo de elegir una revisión entre todas las posibles utilizaremos el criterio utilizado en [Hansson 94] en donde una función de selección s es pensada como la función de selección del elemento menos valioso de cada explicación. Es por lo tanto razonable que s esté basada sobre una relación binaria que representa un valor epistémico comparativo. La condición necesaria que debe cumplir esta relación es que sea acíclica:

Definición 2.10 *Sea* Π *un programa y* $\prec$ *una relación sobre los elementos de* Π. *Entonces* $\prec$ *es* acíclica *si y solo si para todo entero positivo* n: *si* $\{L_1, \ldots, L_n\}$ *son elementos de* Π, *entonces no ocurre que* $L_1 \prec L_2 \prec \ldots \prec L_n \prec L_1$.

En [Hansson 94] se especifica cómo una función de selección elige los elementos menos valiosos de un kernel. En el caso de programas con negación por falla toda función de selección basada en esta información epistémica también seleccionará de cada soporte revisable aquellos elementos minimales según la relación $\prec$ como definimos a continuación.

Definición 2.11 *Una función de selección* s *para* Π basada *sobre una relación* $\prec$ *se define de la siguiente manera:*
$s(\langle I, O, L \rangle) = \langle I', O', L \rangle$ *donde:*
$x \in I'$ *si y solo si* $x \in I$ *y no existen* $y \in I$ *y* $z \in O$ *tal que* $y \prec x$ *o not* $z \prec x$ *y*
$x \in O'$ *si y solo si* $x \in O$ *y no existen* $y \in I$ *y* $z \in O$ *tal que* $y \prec x$ *o not* $z \prec x$

En la definición siguiente establecemos cómo una función de selección determina una revisión para la operación de contracción que definiremos en la siguiente sección.

Definición 2.12 *Sea* Π *un programa contraíble por* L*, y* s *función de selección, una* revisión basada en s *es un triple* $\langle I, O, L\rangle$ *tal que*
$I = \bigcup\{I' : s(e) = \langle I', O', L\rangle$ *y* e *es un soporte revisable de* $\Pi\}$
$O = \bigcup\{O' : s(e) = \langle I', O', L\rangle$ *y* e *es un soporte revisable de* $\Pi\}$

Ejemplo 6 *Si tomamos los soportes revisables del ejemplo anterior,* $\langle\{\}, \{q, c\}, p\rangle$ *es una revision de* Π*, para el caso de* $Rev(\Pi) = \langle\{a\}, \{q, c\}\rangle$*.*

2.4 Operador de Contracción

Para un programa Π y un literal L que pertenece a $Cn(\Pi)$ debemos definir un operador de contracción $\div$ de manera que $\Pi \div L$ sea un nuevo programa cuyo significado no contenga a L. En la teoría de cambio de creencias esta propiedad del operador es llamada el postulado de éxito. Es decir

$$L \notin Cn(\Pi \div L)$$

Otra propiedad deseable del operador de contracción es la de *inclusión* que postula que la contracción no debe agregar nuevos elementos a la teoría. En el caso de los programas lógicos, dicha propiedad sería

$$Cn(\Pi \div L) \subseteq Cn(\Pi).$$

Otros postulados establecen la minimalidad del cambio en el sentido de cambiar lo menos posible de la teoría original con el fin de lograr el éxito de la contracción. Para los programas lógicos, se deben considerar dos casos, según si el significado de un programa está basado en una semántica tri-valuada o bi-valuada. En el caso de una semántica tri-valuada, la operación de contracción puede hacer que le valor de un literal pase de verdadero a indefinido, mas que de verdadero a falso como en el caso bi-valuado. Para realizar estos cambios mínimos, se pueden utilizar reglas de inhibición que prohiben la falsedad de un átomo en algún modelo. En el caso de semánticas bi-valuadas, el carácter no monótono de la negación por falla, hace que la propiedad de inclusión no se cumpla. Para definir un operador de contracción bi-valuado, que cumpla la condición de éxito, utilizaremos el siguiente procedimiento. Dado un programa Π y un literal L se inhabilitan todas las explicaciones con consecuente L , obteniendo un nuevo programa Γ. Si Γ no contiene en su significado a L, la operación de contracción estará cumplida. Puede ocurrir sin embargo que nuevas explicaciones surjan para L, en ese caso se vuelve a obtener una revisión y se modifica el programa, hasta que la condición de éxito sea alcanzada o hasta que el programa no pueda ser contraído. Cada una de las revisiones debe estar basadas en los revisables del programa original, es decir en $Rev(\Pi)$ garantizando que el procedimiento termine. El procedimiento es el siguiente:

Definición 2.13 *El resultado de la operación de contracción bivaluada* $\Pi \div L$ *se obtiene de la siguiente manera:*

Dados Π *un programa y* L *un literal clásico.*
Si Π *es contraíble por* L *, sea* $\langle I, O, L\rangle$ *una revisión basada en* $Rev(\Pi)$
obtengo un nuevo programa $\Gamma = (\Pi - I) \cup O$*.*
Si $L \notin Cn(\Gamma)$*, retorno* Γ
en caso contrario repito lo siguiente hasta que $L \notin Cn(\Gamma)$*:*
si Γ *no es contraíble por* L *termino diciendo que* Π *no es contraíble.*
en caso contrario sea $\langle I, O, L\rangle$ *una revision de* Γ *basada en* $Rev(\Pi)$*,*
a Γ *le asigno un nuevo programa* $(\Gamma - I) \cup O$

En cada paso el programa nuevo se obtiene retirando las reglas del componente I y agregando como hechos los elementos del componente O. Esto produce que se eliminen todas las explicaciones encontradas hasta entonces. Si surgen nuevas, el procedimiento continúa.

Ejemplo 7 *En nuestro ejemplo, obtenida la revisión* $\langle\{\}, \{q, c\}, p\rangle$, $\Pi \div p$ *es el programa:*

$$p \leftarrow \neg r, not\ q.$$
$$p \leftarrow not\ b.$$
$$\neg r \leftarrow a.$$
$$b \leftarrow q, c.$$
$$a.$$
$$q.$$
$$c.$$

el significado de este programa es $Cn(\Pi \div p) = \{a, q, c, b, \neg r, not\ p\}$

En este ejemplo, después de realizar la primera revisión, no surgieron nuevas explicaciones para p, por lo tanto no fue necesario repetir el procedimiento. En la sección siguiente veremos un ejemplo aplicado a debugging donde la condición de éxito no se logra en una primer instancia y es necesario combatir nuevas explicaciones que surgen por el hecho de eliminar suposiciones producidas por la negación por falla.

3 Debugging por medio de Contracción de Creencias

En esta sección aplicaremos la operación de revisión de programas con el fin de implementar un mecanismo de debugging para programas lógicos.

En [Lloyd 87] la definición de programa incorrecto y los tipos de errores, está basada sobre programas esquemáticos y con la completación de predicados como semántica. Los programas sobre los cuales definimos la operación de revisión de creencias son programas no esquemáticos y cuya semántica está definida por el operador de consecuencia. Por lo tanto necesitamos reformular las definiciones y las proposiciones de [Lloyd 87] para nuestro caso particular.

Definición 3.1 *Sea* Π *un programa, una* interpretación pretendida M *para* Π *es un conjunto de elementos de regla con literales en Lit*

Dada una interpretación pretendida, definida por el Oráculo, la correctitud de un programa depende de la correspondencia entre esa interpretación y las consecuencias del programa.

Definición 3.2 *Sea* Π *un programa y* M *una interpretación pretendida para* Π. *Decimos que* Π *es* correcto *con respecto a* M *si* $Cn(\Pi) \subseteq M$ *y es* completo *con respecto a* M *si* $M \subseteq Cn(\Pi)$.

Los errores en los programas lógicos pueden manifestarse a través de dos clases de síntomas. Si L es un literal tal que $\models L$ es derivable en el cálculo SLDNF para un programa Π, y no pertenece a la interpretación pretendida entonces es una solución errónea para el programa con respecto a la interpretación. Del mismo modo si $=|G$ es derivable en el cálculo SLDNF y pertenece a la interpretación constituye una solución faltante para el programa con respecto a la interpretación. Formalmente:

Definición 3.3 *Si* Π *es un programa,* M *una interpretación pretendida y* L *un literal en Lit.*

- *Si* L *es exitoso relativo a* Π, *y* $L \notin M$. L *es una solución errónea para* Π *con respecto a* M.

- *Si L falla relativo a Π y $L \in M$, entonces L es una solución faltante para Π con respecto a M*

Existen dos tipos de errores que producen este mal funcionamiento: literales no cubiertos y reglas incorrectas. Las definiremos de la siguiente manera:

Definición 3.4 *Sea Π un programa lógico y M una interpretación pretendida, entonces L es un literal no cubierto si $L \in M$ y no existe ninguna regla $L \leftarrow Cuerpo$ en Π tal que $Cuerpo \subseteq M$.*

Definición 3.5 *Sea Π un programa lógico y M una interpretación pretendida, entonces una regla $Cabeza \leftarrow Cuerpo$ es una regla incorrecta en Π si $Cabeza \notin M$ y $Cuerpo \subseteq M$.*

Un programa es incorrecto con respecto a una interpretación pretendida cuando existe un literal no cubierto o cuando existe una regla incorrecta.

Supongamos el siguiente ejemplo de programa que expresa que a mi esposa le gusta un regalo siempre que éste no sea demasiado caro, y que dicho regalo es caro si yo no lo puedo comprar.

Ejemplo 8 *Sea Π el siguiente programa:*

$$le\_gusta \leftarrow not\ es\_caro.$$
$$es\_caro \leftarrow not\ puedo\_comprar.$$

Las consecuencias de este programa según la definición 2.1 es:

$$Cn(\Pi) = \{not\ le\_gusta, es\_caro, not\ puedo\_comprar\}$$

es decir que no existe nada que diga que el regalo le gusta a mi esposa, éste es caro y yo no lo puedo comprar. Supongamos sin embargo que el significado pretendido con el programa sea que aún cuando no le gusta, el regalo no es caro y yo lo puedo comprar. Es decir que

$$M = \{not\ le\_gusta, not\ es\_caro, puedo\_comprar\}$$

Definitivamente $le\_gusta \leftarrow not\ es\_caro.$ es una regla incorrecta ya que

$$\{not\ es\_caro\} \subseteq M$$

y $le\_gusta \notin M$. También se puede observar que *puedo_comprar* es un literal no cubierto en el programa con respecto a M, pues ninguna regla lo tiene es su cabeza.

El procedimiento de debugging que definiremos no estará involucrado en localizar las reglas incorrectas o los literales no cubiertos, pero el programa resultante no contendrá las reglas incorrectas del programa original y tendrá soporte para los literales no cubiertos del programa erróneo.

Dado un programa Π y un significado pretendido M, el procedimiento obtendrá un programa Π_M de manera que Π_M sea un programa correcto y completo con respecto a M. El proceso de debugging cuenta de tres etapas como lo muestra la figura 1

El primer paso es producir un programa al que se le agrega la información provista por el significado pretendido por medio de lo que llamaremos *reglas de oráculo.*

Definición 3.6 *Sea* incorrecto *un nuevo literal que pertenezca a Lit. Dado M un significado pretendido, entonces una* regla de oráculo *es una regla incorrecto $\leftarrow Comp(A)$ donde A es un elemento de regla que pertenece a M.*

Ejemplo 9 *Dado $M = \{not\ le\_gusta, not\ es\_caro, puedo\_comprar\}$ entonces las siguientes son reglas de oráculo:*

$$incorrecto \leftarrow le\_gusta$$
$$incorrecto \leftarrow es\_caro$$
$$incorrecto \leftarrow not\ puedo\_comprar$$

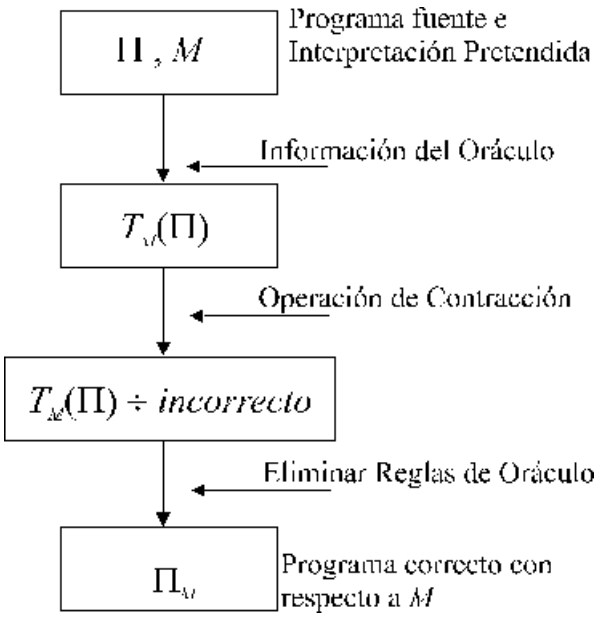

Figura 1: Etapas del proceso de debugging

El programa resultante de agregar a un programa Π las reglas de oráculo definidas por el significado pretendido M será llamado $T_M(\Pi)$.

Definición 3.7 *Dado un programa* Π*, y* M *su significado pretendido, entonces* $T_M(\Pi)$ *es el programa que se obtiene al agregar a* Π *las siguientes reglas:*

- *para cada literal clásico* $L \in M$ *la regla* $incorrecto \leftarrow not\ L$.
- *para cada elemento de regla not* $A \in M$ *la regla* $incorrecto \leftarrow A$.

Dado que *incorrecto* es un nuevo átomo que no pertenece al lenguaje y $T_M(\Pi)$ y Π solo difieren en reglas con *incorrecto* en la cabeza, las consecuencias de $T_M(\Pi)$ y Π solo pueden diferir en *incorrecto*

Ejemplo 10 *Dados* Π *y* M *del ejemplo 8, el programa* $T_M(\Pi)$ *es el siguiente:*

$$le\_gusta \leftarrow not\ es\_caro.$$
$$es\_caro \leftarrow not\ puedo\_comprar.$$
$$incorrecto \leftarrow le\_gusta$$
$$incorrecto \leftarrow es\_caro$$
$$incorrecto \leftarrow not\ puedo\_comprar$$

Si *incorrecto* es una consecuencia de $T_M(\Pi)$ entonces quiere decir que existen soluciones faltantes o soluciones falsas para el programa Π. La segunda etapa entonces es contraer el programa por el literal *incorrecto* mediante la operación de contracción definida en la sección anterior donde los revisables del programa no contengan las reglas de oráculo.

El programa resultante de la operación de contracción, por la condición de éxito de la operación no inferirá *incorrecto*. De hecho, las consecuencias del programa contraído coincidirán con el significado pretendido como lo muestra el siguiente resultado.

Proposición 3.8 *Si* Π *es un programa y* M *es su significado pretendido, entonces*

$$Cn(T_M(\Pi) \div incorrecto) = M$$

Prueba:

Sea $A \in Cn(T_M(\Pi) \div incorrecto)$. Existen dos casos, A puede ser un literal L o un elemento de regla $not\ L$. En el primer caso $A = L$, supongamos $L \notin M$, como M provee una información completa, el elemento de regla $not\ L \in M$, luego por la definición 3.7 $incorrecto \leftarrow L \in T_M(\Pi)$ como $incorrecto \leftarrow L \notin \Pi$, $incorrecto \leftarrow L$ no pertenece a los revisables de $T_M(\Pi)$. Entonces $incorrecto \leftarrow L \in T_M(\Pi) \div incorrecto$.

Dado que $L \in Cn(T_M(\Pi) \div incorrecto)$ por la proposición 2.4 existe una explicación $\langle I, O, L\rangle$ en $T_M(\Pi) \div incorrecto$, luego por la definición 2.3

$$\langle \{incorrecto \leftarrow L\} \cup I, O, incorrecto\rangle$$

es una explicación de $T_M(\Pi) \div incorrecto$, lo que viola la condición de éxito del operador $\div$ por lo tanto $L \in M$.

En el caso de que $A = not\ L$, la prueba es similar, si suponemos $not\ L \notin M$, entonces $L \in M$ dada la información completa de M, entonces por la definición 3.7 $incorrecto \leftarrow not\ L \in T_M(\Pi)$, como no es una regla revisable al no pertenecer a Π, $incorrecto \leftarrow not\ L \in T_M(\Pi) \div incorrecto$. Dado que $not\ L \in Cn(T_M(\Pi) \div incorrecto)$ por la proposición 2.4 existe una explicación $\langle I, O, not\ L\rangle$ en $T_M(\Pi) \div incorrecto$, luego por la definición 2.3

$$\langle \{incorrecto \leftarrow not\ L\} \cup I, O, incorrecto\rangle$$

es una explicación de $T_M(\Pi) \div incorrecto$, lo que viola la condición de éxito del operador $\div$ por lo tanto $not\ L \in M$.

Recíprocamente, Sea $A \in M$, A puede ser un literal L o un elemento de regla $not\ L$, en el primer caso supongamos que $L \notin Cn(T_M(\Pi) \div incorrecto)$.

Si $L \in M$ entonces por la definición 3.7 $incorrecto \leftarrow not\ L \in T_M(\Pi)$ como $incorrecto \leftarrow not\ L \notin \Pi$, $incorrecto \leftarrow not\ L$ no pertenece a los revisables de $T_M(\Pi)$. Entonces $incorrecto \leftarrow not\ L \in T_M(\Pi) \div incorrecto$. Como supusimos $L \notin Cn(T_M(\Pi) \div incorrecto)$ por la proposición 2.4 existe una explicación $\langle I, O, not\ L\rangle$ en $T_M(\Pi) \div incorrecto$, luego por la definición 2.3 de explicación,

$$\langle \{incorrecto \leftarrow not\ L\} \cup I, O, incorrecto\rangle$$

es una explicación de $T_M(\Pi) \div incorrecto$, lo que viola la condición de éxito del operador $\div$. Por lo tanto $L \in Cn(T_M(\Pi) \div incorrecto)$. En el caso de que $A = not\ L$, la prueba es similar.

Retomemos el programa del ejemplo 10 y apliquemos la definición 2.13 con el fin de obtener $T_M(\Pi) \div incorrecto$. Como se utilizará la operación de contracción bivaluada se debe tener en cuenta que nuevas explicaciones pueden surgir para el literal *incorrecto*. Los revisables en la operación son las reglas que originalmente se encontraban en Π y la jerarquía entre los elementos del programa que se utilizará tomará a las reglas con mayor valor epistémico que las suposiciones de la negación por falla.

De acuerdo a la definición 2.3 las únicas explicaciones con consecuente *incorrecto* en Π son:

$\langle \{incorrecto \leftarrow es\_caro, es\_caro \leftarrow not\ puedo\_comprar\}, \{puedo\_comprar\}, incorrecto\rangle$
$\langle \{incorrecto \leftarrow not\ puedo\_comprar\}, \{puedo\_comprar\}, incorrecto\rangle$

El conjunto de revisables de la operación es $Rev(T_M(\Pi)) = \langle \Pi, \{puedo\_comprar\}\rangle$, por lo tanto los soportes revisables con consecuente *incorrecto* de Π son:

$\langle\{es\_caro \leftarrow not\ puedo\_comprar\}, \{puedo\_comprar\}, incorrecto\rangle$
$\langle\{\}, \{puedo\_comprar\}, incorrecto\rangle$

Dado que $not\ puedo\_comprar \prec es\_caro \leftarrow not\ puedo\_comprar$ la función de selección s basada en $\prec$ no selecciona a $es\_caro \leftarrow not\ puedo\_comprar$ por no ser minimal, así tenemos:

$$\begin{aligned} s(\langle\{es\_caro \leftarrow not\ puedo\_comprar\}, \{puedo\_comprar\}, incorrecto\rangle) &= \\ \langle\{\}, \{puedo\_comprar\}, incorrecto\rangle \text{ y} & \\ s(\langle\{\}, \{puedo\_comprar\}, incorrecto\rangle) &= \\ \langle\{\}, \{puedo\_comprar\}, incorrecto\rangle & \end{aligned}$$

De esta manera, por la definición 2.12 $\langle\{\}, \{puedo\_comprar\}, incorrecto\rangle$ es una revisión basada en la función de selección s. Por lo tanto el procedimiento de la definición 2.13 obtiene temporalmente el programa Γ igual a:

$le\_gusta \leftarrow not\ es\_caro.$
$es\_caro \leftarrow not\ puedo\_comprar.$
$incorrecto \leftarrow le\_gusta$
$incorrecto \leftarrow es\_caro$
$incorrecto \leftarrow not\ puedo\_comprar$
$puedo\_comprar.$

Este programa todavía infiere incorrecto, entonces obtenemos una vez más las explicaciones con consecuente *incorrecto*. Por la definición 2.3 la única explicación con consecuente *incorrecto* en Γ es:

$$\langle\{incorrecto \leftarrow le\_gusta, le\_gusta \leftarrow not\ es\_caro, puedo\_comprar\}, \{es\_caro\}, incorrecto\rangle$$

Nuevamente, teniendo en cuenta que $Rev(\Pi) = \langle\{le\_gusta \leftarrow not\ es\_caro, es\_caro \leftarrow not\ puedo\_comprar\}, \{puedo\_comprar\}\rangle$, de acuerdo a la definición 2.8 el único soporte revisable con consecuente *incorrecto* de Γ es:

$$\langle\{le\_gusta \leftarrow not\ es\_caro\}, \{\}, incorrecto\rangle$$

entonces

$$s(\langle\{le\_gusta \leftarrow not\ es\_caro\}, \{\}, incorrecto\rangle) = \langle\{le\_gusta \leftarrow not\ es\_caro\}, \{\}, incorrecto\rangle$$

es una revisión basada en s de Γ. El procedimiento obtiene entonces el programa Γ:

$es\_caro \leftarrow not\ puedo\_comprar.$
$incorrecto \leftarrow le\_gusta$
$incorrecto \leftarrow es\_caro$
$incorrecto \leftarrow not\ puedo\_comprar$
$puedo\_comprar.$

que no infiere *incorrecto*, por lo tanto la condición de éxito es lograda y $T_M(\Pi) \div incorrecto = \Gamma$.

El último paso en el proceso de debugging es eliminar las reglas de oráculo utilizadas durante el proceso.

Definición 3.9 *Dado un programa* Π *y* M *su significado pretendido entonces el programa* Π *de acuerdo a* M *que notaremos* Π_M *se obtiene de* $T_M(\Pi) \div incorrecto$ *eliminando todas las reglas de oráculo.*

El programa obtenido Π_M es correcto y completo con respecto al significado M.

Proposición 3.10 *Dado un programa* Π *y su significado pretendido* M *entonces* Π_M *es correcto y completo con respecto a* M

Prueba:

De la proposición 3.8 surge que $T_M(\Pi) \div incorrecto$ es correcto y completo con respecto a M, y dado que $incorrecto \notin Cn(T_M(\Pi) \div incorrecto)$, las reglas con incorrecto en la cabeza pueden ser eliminadas sin variar las consecuencias del programa, por lo tanto Π_M es también correcto y completo.

Ejemplo 11 *Si de* $T_M(\Pi) \div incorrecto$ *del ejemplo anterior se eliminan las reglas de oráculo se obtiene* Π_M *igual a:*

$$es\_caro \leftarrow not\ puedo\_comprar.$$
$$puedo\_comprar$$

El conjunto de consecuencias de este programa es

$$Cn(\Pi_M) = \{not\ le\_gusta, not\ es\_caro, puedo\_comprar\}$$

que coincide con M.

4 Conclusiones y Trabajo Futuro

Los procesos de debugging declarativo tradicionales comienzan con el diagnóstico de una solución faltante o una solución errónea y a partir de allí los algoritmos inician una sesión interactiva con el usuario a fin de consultar el significado pretendido a medida de que recorre los árboles de prueba con el fin de encontrar la causa del error.

El proceso de debugging presentado en la sección anterior por el contrario toma la opción de que el programador presente el programa y el significado pretendido en un principio y que luego el sistema se encargue de hacerlos compatibles por medio del proceso de debugging. El sistema no necesita más interacción con el usuario debido que descansa sobre el mecanismo de revisión de creencias.

Las reglas del oráculo que introducen la información del significado pretendido pueden ser de una forma más general que las utilizadas en este trabajo. Por ejemplo si en el significado pretendido para un programa se considera a los literales A o B verdaderos, pero no a ambos. Una revisión del programa podría hacerse introduciendo la siguiente regla de oráculo: $incorrecto \leftarrow A, B$. Otro aspecto no explorado es la posibilidad de que el conjunto de reglas de oráculos involucradas represente información parcial del significado pretendido. Hemos asumido que el oráculo tiene información completa sobre el valor pretendido para cada literal, en la práctica sin embargo sería útil que el usuario tuviera la posibilidad de hacer compatible un programa con un significado pretendido que no tuviera una total información de cada literal.

Otra generalización necesaria es definir las operaciones de contracción de creencias sobre programas esquemáticos, permitiendo diferenciar entre esquema de regla e instancia, y así poder aplicar un proceso de debugging basado en revisión de creencias en los programas de PROLOG usuales.

Referencias

[Alchourrón 85] Alchourrón, C. E. and Makinson, D. (1985). On the Logic of Theory Change: Safe Contraction. *Studia Logica*, 44:405–422.

[Calejo 91] Calejo, M., Pereira, M., and Moniz, L. (1991). Declarative source debugging. In *Lecture Notes in Computer Science*, volume 541, pages 237–249. Springer-Verlag.

[Dershowitz 87] Dershowitz, N. and Jeng Lee, Y. (1987). Deductive debugging. In *Proceedings of the 1987 Symposium Logic Programming.*

[Ducassé 93] Ducassé, M. (1993). A pragmatic survey of automated debugging. In *Lecture Notes in Computer Science*, volume 749, pages 1–15. Springer-Verlag.

[Gelfond 88] Gelfond, M. and Lifschitz, V. (1988). The Stable model semantics for logic programming. In Kowalski, R. and Bowen, K. A., editors, *5th Int. Conf. on LP*, pages 1070–1080. MIT Press.

[Gelfond 90] Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *ICLP'90*, pages 579–597.

[Hansson 94] Hansson, S. O. (1994). Kernel Contraction. *Journal of Simbolic Logic*, 59:845–859.

[Lloyd 87] Lloyd, J. W. (1987). Declarative error diagnosis. *New Generation Computing*, 5:133–154.

[Pereira 86] Pereira, L. M. (1986). Rational Debugging in Logic Programming. In *Thirth Int. Conf. Logic Programming.*

[Pereira 92] Pereira, L. M. and Alferes, J. J. (1992). Well founded semantics for logic programs with explicit negation. In Neumann, B. ., editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons.

[Przymusinski 90] Przymusinski, T. (1990). Extended stable semantics for normal and disjunctive programs. In *ICLP'90*, pages 459–477.

[Shapiro 82] Shapiro, E. Y. (1982). *Algorithmic Program Debugging.* PhD thesis, Yale University.

[Van Gelder 91] Van Gelder, A., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

[Vaucheret 98] Vaucheret, C. A. and Simari, G. R. (1998). Un operador general de contracción para programas lógicos. In *V Workshop de Aspectos Teóricos de Inteligencia Artificial*, Neuquén.